The Implementation of iData
A Case Study in Generic Programming

Rinus Plasmeijer and Peter Achten

Software Technology, Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen {rinus, P.Achten}@cs.ru.nl

Abstract. The iData Toolkit is a toolkit that allows programmers to create interactive, type-safe, dynamic web applications with state on a high level of abstraction. The key element of this toolkit is the iData element. An iData element is a form that is generated automatically from a type definition and that can be plugged in in the web page of a web application. In this paper we show how this automatic generation of forms has been implemented. The technique relies essentially on *generic programming*. It has resulted in a concise and flexible implementation. The kernel of the implementation can be reused for any graphical package. The iData Toolkit is an excellent demonstration of the expressive power of modern generic (poly-typical) programming techniques.

1 Introduction

In this paper we present the implementation of the iData Toolkit, which is a novel toolkit to program *forms* in dynamic server-side web applications. The low level view, and standard definition, of a form is that of a collection of (primitive) interactive elements, such as text input fields, check boxes, radio buttons, pull down menus, that provide the application user with a means to exchange structured information with the web application. Seen from this point of view, and if programmed that way, creating forms results in a lot of low level HTML coding. A high level view of forms is to think of them as being *editors of structured values* of appropriate type. From the type, the low level realization can be derived automatically. This can be done once by the toolkit developer. Seen from that point of view, and if programmed that way, creating forms is all about creating data types. This results in a lot less code plumbing and no HTML-coding at all.

In the iData Toolkit project, we have adopted the high level view of forms described above. We call these high level forms iData. An iData has two major components: **(i)** a *state*, or *value*, which type is determined by the programmer, and **(ii)** a *form*, or *rendering*, which is derived automatically by the toolkit from the state and its type. The programmer manipulates the iData in terms of the state and its type, whereas the user manipulates the iData in terms of a low-level form. Clearly, the iData Toolkit needs to mediate between these two worlds: every possible type domain must be mapped to forms, and every user action on these forms must be mapped back to the original type domain, with a possibly different value. This is the challenge that is addressed in this paper.

An approach as sketched above can be implemented in any programming language with good support for data types and type-driven programming. Modern functional programming languages such as Clean [22, 2] and Haskell [19] come with highly expressive type systems. One example of type-driven programming is *generic programming*[12, 13, 1], which has been incorporated in Clean and GenericH∀skell [16]. In this paper we use Clean. We assume the reader is familiar with functional and generic programming.

Generic programming has proven productive in the iData Toolkit by providing us with concise and flexible solutions for many of the chores of web programming. In this paper we focus on its crucial contribution to solving the main challenge in the context of the iData Toolkit: the automatic creation of forms from arbitrary data types, and the automatic creation of the correct data type and value from a user-manipulated form. The key idea is that each iData is fully responsible for keeping track of its rendering, its state recovery, and correctly handling user-edit operations. They, and only they, can do this because they have all type information that is necessary for these operations.

It should be observed that although we give a few examples, this paper is about the implementation of the iData Toolkit. Due to limitations of space, we cannot explain the programming method. This is presented elsewhere [20, 21]. We have used the iData Toolkit to create realistic real world applications, such as a web shop. These demonstrate that this technique can be used in practice.

This paper is structured as follows. We first introduce the concept and implementation challenges of iData (Sect. 2). Then we present the concrete implementation of iData (Sect. 3). After this, we discuss the achieved results (Sect. 4). We present related work (Sect. 5) and conclude (Sect. 6).

2 The concept of iData

In this section we explain the main concepts of the iData Toolkit by means of a few key toy examples (Sect. 2.2–2.6). They illustrate the implementation challenges that need to be solved. These are summarized in Sect. 2.7. Please notice that although the code of these examples has a static flavor, each of these examples are complete interactive web applications. We first present the major design decisions in Sect. 2.1.

2.1 Major Design Decisions

The key idea of an iData Toolkit program is that it is a function that computes an HTML page that contains a set of interconnected iData elements. An iData element is a self contained interactive element that is in charge of its state. The state can be any programmer defined data type. The iData Toolkit is constructed in such a way that the state of an iData element is always a valid instance of the type. Type constraints on the input are not always sufficient: individual iData elements can impose additional constraints on the values of their state that can not be expressed with types, or they are interconnected and need to

modify their state as a consequence of the modification of another iData change. For this reason, every *complete*[1] user manipulation of an iData element requires a response of that element and the iData elements that are connected with it. Currently, this has been implemented by enforcing a round trip between browser and server.

The code of an iData Toolkit application is a single function that is evaluated every time a client web browser requires a web page from this application. Initially, no previous states are available to the application, and the iData elements are activated with their initial values. During subsequent requests, the web browser provides the states of all iData elements and detailed information about which iData element has been modified by the user. The implementation of the toolkit uses this information to recover previous unaltered states, and create a valid altered new state. This is hidden completely from the programmer. He can reason about the program as a collection of interconnected iData elements, one of which has a modified state value.

Generic programming has been crucial to implement the core functionality of the iData Toolkit: rendering state in terms of low-level forms, recovering previous states, and incorporating arbitrary user modifications in states. Generic programming has also been used for tasks that could have been done with more traditional means: (de-)serialization of states, and printing HTML. A key advantage of generic programming is that one has a default application for free for any type. If this generic solution is not appropriate, the programmer (or toolkit developer) can use *specialization* to replace the default solution with a more suitable solution. Specialization can be done for individual iData elements, but also for complete types. With specialization, the iData Toolkit can be extended with elements that have more logic at the client side, for instance for specialized text input parsers.

The number and types of iData elements in an HTML page that is generated by an iData Toolkit application depends on the values of the states of its iData elements. During evaluation of the application, these iData elements are activated and need to recover their previous, or altered, state from this collection of states. This requires an identification mechanism that is able to associate typed iData elements with serialized states. Exceptional cases are the absence of such a state (initial occurrence of an iData element), or that the serialized state is of incorrect type (page originated from a different application). The problem is reminiscent of manipulating typed content that comes from files. Currently, this has been implemented pragmatically: iData elements are identified with text labels. The state of an iData element can be recovered successfully if it is present in the set of previous states and can be converted successfully to a value of its type. In every other case, the iData element obtains the initialization value with which it is associated in the program code. This makes the approach robust. We are well aware that this is a deficient solution, particularly considering the strongly typed discipline that we advocate with the iData Toolkit.

[1] In a text box this is the completion of the input, either by changing the input focus or - for single line edit boxes - pressing the enter key.

The code below shows the standard overhead of every iData Toolkit program:

```
module IFL2005Examples
import StdEnv, StdHtml                                          1.

Start :: *World → *World                                        2.
Start world = doHtml example world                              3.
```

The proper library modules need to be imported (line 1). Lines 2–3 declare the main function of every Clean program. The *uniqueness attribute* `*` just in front of `World` guarantees that values of this type are always used in a *single threaded manner*. Clean uses *uniqueness typing* [5, 6] to allow destructive updates and side-effects. The opaque type `World` represents the entire external environment of the program. The iData program is given by the function `example :: *HSt → (Html,*HSt)`. The wrapper function `doHtml` turns this function into a common Clean program. It initializes the `HSt` value with all serialized values that can be found in the HTML page, and includes the `World` as well. This implies that every iData Toolkit application has full access to the external world, and can, for instance, connect to databases and so on. Below, we only show the `example` functions, and skip the standard overhead.

2.2 iData Have Form

The first example demonstrates the fact that iData elements are type driven. It creates a simple `Int` editor (Fig. 1**(a)**).

```
example :: *HSt → (Html,*HSt)                                   1.
example hst                                                     2.
    # (nrF,hst) = mkEdit (nIDataId "nr") 1 hst                  3.
    = mkHtml "Int editor" [ H1 [] "Int editor", BodyTag nrF.form ] hst   4.
```

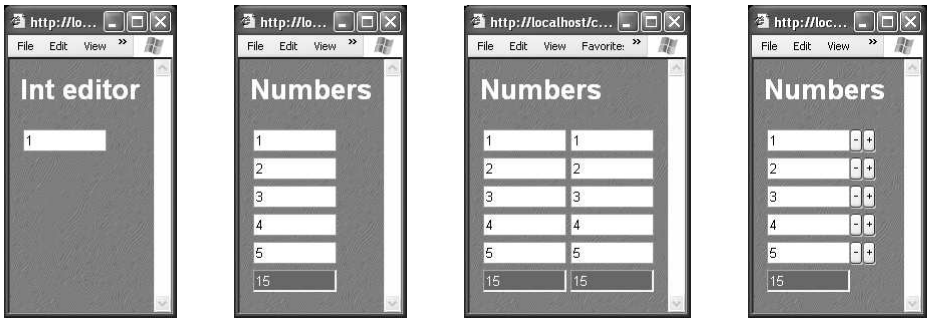

Fig. 1. Key toy examples: **(a)** a simple integer iData, **(b)** summing the value of iData, **(c)** sharing iData, and **(d)** model-view separation of iData.

Passing multiple environments around explicitly is supported syntactically in Clean by means of ♯-definitions. These are non-recursive *let*-definitions, which scope extends to the bottom, but not the right-hand side. This is the standard approach in Clean. Even though the examples in this paper do not exploit the flexibility of multiple environment passing (by for instance connnecting to a database system), we present them in this style.

Key features of the iData Toolkit that are illustrated in this small example are the activation of an iData element, `nrF`, from an initial value and its type, `1::Int`. It is identified throughout the program with the value (`nDataId "nr"`) :: `IDataId`. This is done with the function `mkEdit` (line 3). This iData element has a rendering in terms of a form, `nrF.form` ($r.f$ denotes the selection of field f from record r). The rendering is a text edit box in which only integer denotations can be entered. In general, a user can only enter input that is type-safe.

The definition of the web page, given by the function `mkHtml :: String [BodyTag] *HSt → (Html,*HSt)`, is cleanly separated from the definition of the iData elements. The `[BodyTag]` argument represents the page content. The algebraic type `BodyTag` is discussed in more detail in Sect. 3.5. In these examples, we use its data constructor `H1` which represents the `<h1></h1>` HTML tag, and its data constructor `BodyTag` which turns a list of `BodyTag`s into a single `BodyTag`.

2.3 iData Have Value

In this example we show that iData not only have a form rendering, but also a value in terms of the type that generated them.

```
example hst                                                              1.
    ♯ (nrFs,hst) = seqList [mkEdit (sumId nr) nr \\ nr ← [1..5]] hst     2.
    = mkHtml "Numbers" [ H1 [] "Numbers", sumtable nrFs ] hst            3.

sumtable nrFs = STable [] ([nrF.form \\ nrF ← nrFs]                      4.
                                        ++                               5.
                           [[toHtml (sum [nrF.value \\ nrF ← nrFs])]])   6.
sumId i       = nDataId ("sum"<$i)                                       7.
```

Fig. 1**(b)** shows the result of the above code. Five iData elements are activated: `nrFs :: [IData Int]` (line 2). The function `sumtable` (lines 4-6) places their *forms* in a column, underneath of which the sum of their *values* is displayed. Whenever the user alters one of the iData elements, the new sum is calculated and displayed at the bottom of the iData elements. The reason that this statically looking program has interactive behaviour, is that the behaviour is delegated to each of the iData elements that are activated. This is why we prefer to speak of activation of iData.

The value of an iData is given by the `.value` field of that iData. The library function `toHtml` uses the generic form rendering function to render values of arbitrary type into HTML. The overloaded operator `<$` appends a `String` version of its second argument to its first argument.

2.4 iData Have Sharing

iData elements with the same identification value refer to the same iData element. A first advantage of this scheme is that iData serve as storages of arbitrary types. Hence, we do not need to introduce a separate concept for storing data. A second advantage is that both the value and rendering of iData can be used arbitrarily many times in a HTML page without causing ambiguity problems. We illustrate the latter by replicating the column of integer iData and their sum in the example below (see Fig. 1**(c)**):

```
example hst
  ♯ (nrFs,hst) = seqList [mkEdit (sumId nr) nr \\ nr ← [1..5]] hst
  = mkHtml "Numbers"
       [ H1 [] "Numbers", STable [] [[sumtable nrFs],[sumtable nrFs]] ] hst
```

Editing any of the iData elements also automatically affects the other iData in the same row. The sum is displayed twice, at the bottom of both columns.

2.5 iData Have Model-View Separation

In this example we demonstrate that the type of an iData can be uncoupled from its rendering. The rendering can be derived instead from a different data type, provided that the programmer defines the mapping between these two data types. In this way, the type of the iData serves as its *model*, whereas the rendered data type serves as its *view*. In Sect. 3.1, we explain the mapping and its implementation in detail. Here, we assume the existence of a function, `counterIData`, that has an `Int` model type, but a (`Counter Int`) view type, where `Counter` is defined as a synonym type `:: Counter a :== (a,Button,Button)`.

```
counterIData :: IDataId Int *HSt → (IData Int,*HSt)
```

If we replace `mkEdit` in example 2.3, line 2, with `counterIData` then we obtain a program that displays five counters instead of five integer editors (see Fig. 1**(d)**). The counters are *self contained*. The counter iData ensures that its integer value is incremented/decremented at every corresponding button press. Still, it has an integer interface to the programmer, so the remainder of the program does not change. Self contained iData are fully compositional.

2.6 iData Have Specialization

In this example we show that iData can be specialized, just as generic functions can. Suppose we like the counters in Sect. 2.5 much better than the default integer editors that were used in Sect. 2.2 and 2.3. We need to specialize the generic HTML rendering function `gForm` for the `Int` type. This is done by:

```
gForm{|Int|} iDataId i hst  = specialize asCounter iDataId i hst          1.
where asCounter :: IDataId Int *HSt → (IData Int,*HSt)                    2.
      asCounter iDataId i hst                                             3.
          ♯ (counterF,hst) = counterIData iDataId i hst                   4.
```

```
          = ( { changed    = counterF.changed                          5.
              , value      = fst3 counterF.value                       6.
              , form       = counterF.form }, hst )                    7.
fst3 (x,_,_) = x   // Clean standard library function
```

Function `asCounter` (lines 2-7) defines the specialization using `counterIData` (this function is also defined via the specialization mechanism). Also, `asCounter` is a good example of showing the flexibility of iData programming.

The library function

```
specialize :: (IDataId a *HSt → (IData a,*HSt))
               IDataId a *HSt → (IData a,*HSt) | gUpd{|*|} a
```

is able to 'plug in' the specialization function into any arbitrary other iData structure. Given this specialization of `Int` values, in any place where an iData of an Integer value is needed, a counter iData will be made. In such a setting, the programs 2.2, 2.3, and 2.4 now display self contained counters that behave as expected instead of plain integer editors, without any change in the code of these examples.

2.7 Implementation Challenges

The examples given in this section show that an implementation of the iData Toolkit has to be able to perform the following tasks in a strongly typed programming language context: **(i)** map values of arbitrary types to forms, **(ii)** map edit operations in forms to new values of the given types, **(iii)** handle iData elements as elements with shared value and shared rendering, **(iv)** handle model-view separation correctly, and **(v)** handle specialization correctly. The key idea to solve these challenges is by delegating this functionality to each iData element. The implementation is discussed in the next section.

3 The Implementation of iData

In this section we present the implementation of iData. This is based on a single, pivotal function, `mkIData` which applies a number of generic functions to handle the challenges **(i)** upto **(v)** that were mentioned in Sect. 2.7. Because of its complexity, we split up its discussion. In Sect. 3.1 we focus on `mkIData`, its arguments and results, and the way that it incorporates the model-view separation **(iv)**. In Sect. 3.2 we explain the architecture of the `HSt` environment, in which all iData values are stored **(iii)**. In Sect. 3.3 we discuss all rendering issues of iData. Rendering must be done in such a way that forms are generated from types **(i)**, and that user edit operations are correctly mapped back to values of the source type **(ii)**. In Sect. 3.4 we show how specialization **(v)** uses the framework to nest arbitrarily many iData elements inside each other. Finally, we briefly touch on the issue of emitting proper HTML code in the toolkit in Sect. 3.5.

3.1 The Pivotal `mkIData` Function

The iData Toolkit revolves around a single concept, that of an iData. The toolkit has exactly one function to create iData, `mkIData` with type signature:

```
mkIData :: IDataId m (IBimap m v) *HSt → (IData m,*HSt)
        | gForm{|*|}, gUpd{|*|}, gPrint{|*|}, gParse{|*|} v
```

This function is applied to *four* arguments.

The *first* argument is of type `IDataId`. Values of this type unambiguously identify an iData element. The programmer (carefully) chooses `String` identifiers, which is a typical way of identifying forms in web applications. It is the task of the programmer to use unambiguous names in such a way that every use of (`mkIData id`) refers to the same iData element of some type `m`. `IDataId` values are created with one of the functions {`n`,`s`,`p`}[`d`]`IDataId :: String → IDataId`. The programmer also controls the *life span* and *edit mode* of iData elements with `IDataId` values.

```
:: IDataId  = { id::String, lifespan::LifeSpan, mode::Mode }
:: LifeSpan = Page | Session | Persistent
:: Mode     = Edit | Display
```

The life span of an iData element is determined by {`n`,`s`,`p`}: its value is remembered as long as its page is being viewed (`n`), is stored persistently during a session (`s`), or independently of sessions (`p`). By default, values can be edited in the browser. If they should be displayed only, then any of the {`n`,`s`,`p`}`dIDataId` functions can be used.

The *second* argument of `mkIData` is the initial value of the iData element. It is used only when no iData element with given `IDataId` exists. This happens for instance when the page is viewed for the first time.

The *third* argument of `mkIData` defines the model-view abstraction that has been presented in example 2.5. This allows the application to work with iData that have state values of type `m`, but that are *visualized* by means of values of type `v`. This is a variant of the well-known model(-controller)-view paradigm [15]. What is special about it in this context, is that views are also defined by means of a data type, and hence can be handled generically in exactly the same way! This is clearly expressed in the type signature of `mkIData`, which states that the generic machinery must be available for the view model `v`.

The relation between a model `m` and its view `v` is given by the following collection of functions of type `IBimap m v`:

```
:: IBimap m v = { toView    :: m → Maybe v → v
                , updView   :: Bool → v → v
                , fromView  :: Bool → v → m
                , resetView :: Maybe (v → v) }
```

Model values are transformed to views with `toView`. It can use the previous view value if available. The self contained behavior of an iData element is given by `updView`. Its first argument records if it has been changed by the user. The same argument is passed to the function `fromView` which transforms view values back

to model values. Finally, `resetView` is an optional separate normalization *after* the local behavior function `updView` has been applied.

The function `nextModelView` computes a new model-view pair with these functions in the following way:

```
nextModelView :: (IBimap m v) m (Maybe v) Bool → (m,v)
nextModelView ibm init_m maybe_v changed
   # v = ibm.toView   init_m  maybe_v
   # v = ibm.updView  changed v
   # m = ibm.fromView changed v
   # v = case ibm.resetView of Nothing    = v
                               Just reset = reset v
   = (m,v)
```

This explains how the self contained counters in example 2.5 can be constructed. They use the `updView` function to correctly set their integer value.

The *fourth*, and final, argument of `mkIData` is the `HSt` environment that is used to store all iData values in. This environment is discussed thoroughly in Sect. 3.2. Here, we assume that we have the following access functions available on `HSt` environment values:

```
findIDataValue:: IDataId   *HSt → (Bool,Maybe a,*HSt) | gParse{|*|}, gUpd{|*|} a
replaceState  :: IDataId a *HSt → *HSt                | gPrint{|*|} a
resetCount    ::           *HSt → *HSt
```

The function `findIDataValue` locates the stored state of the identified iData element. The boolean result indicates whether this value has been edited by the application user. It uses the generic functions `gParse` and `gUpd` for deserialization purposes and updating values that may have been altered by the user. New iData values are stored in the `HSt` environment with the function `replaceState`. Because these values are serialized, they require the generic function `gPrint`. Finally, `resetCount` makes sure that the internal counting mechanism of the `HSt` environment is reset to zero. The reason for this is also explained in Sect. 3.2.

When applied to the arguments described above, `mkIData` activates the indicated iData element. As a result, it returns a modified `HSt` environment, and an (`IData m`) record value. This record holds the *form* rendering of the iData element, its *value*, as has been discussed in examples 2.2 and 2.3, and the boolean that states iff the iData element has been altered.

```
:: IData m = { form::[BodyTag], value::m, changed::Bool }
```

We can now explain what `mkIData` does with model values of type `m` and view values of type `v`. We walk through its implementation:

```
mkIData :: IDataId m (IBimap m v) *HSt → (IData m,*HSt)
         | gForm{|*|}, gUpd{|*|}, gPrint{|*|}, gParse{|*|} v
mkIData iDataId init_m ibm hst = nextIData (findIDataValue iDataId hst)      1.
where nextIData (changed,maybe_v,hst)
         # (m,v)         = nextModelView ibm init_m maybe_v changed          2.
         # (iData_v,hst) = gForm{|*|} iDataId v (resetCount hst)             3.
         | iData_v.changed && not changed                                    4.
```

```
                    = nextIData (True,Just iData_v.value,hst)          5.
  | otherwise
    # hst           = replaceState iDataId iData_v.value hst           6.
    # iData_m       = {changed=changed,value=m,form=iData_v.form}      7.
    = (iData_m,resetCount hSt)
```

First the possibly modified value of the given iData element is retrieved (line 1), using the `HSt` access function `findIDataValue` that was introduced above. This is a view value, and hence has type `v`. From this value, new model and view values need to be computed (line 2). Next, the view value is rendered (line 3), using the generic rendering function `gForm` (Sect. 3.3). As we have seen in example 2.6, `gForm` can be specialized. With specialization, the programmer *nests* iData inside each other. It may be the case that one of these nested iData has been altered by the user. Due to recursion, its altered value shows up at this level. If this occurs (condition on line 4 holds), then `mkIData` should proceed with the altered value (line 5). In the end, the value of the resulting view iData is stored in the `HSt` environment (line 6). The final iData has as value the new model value that was computed by `nextModelView`, but as rendering the view rendering (line 7).

The function `mkIData` is a powerful tool to create model-view abstractions with. Frequently occurring patterns of this function have been captured with wrapper functions. Consider the `mkEdit` function that we have used in examples 2.2 and 2.3. It can be used as a 'store' in `Display` mode, or as a straight editor in `Edit` mode.

```
mkEdit :: IDataId m *HSt → (IData m,*HSt)
        | gForm{|*|}, gUpd{|*|}, gPrint{|*|}, gParse{|*|} m
mkEdit iDataId=:{mode} m hst
   = mkIData iDataId m
        { toView    = λnew old → case old of (Just v) → v; _ → new
        , updView   = case mode of Edit → λ_ v → v; Display → λ_ _ → m
        , fromView  = λ_ v → v
        , resetView = Nothing } hst
```

3.2 The Implementation of `HSt`

The `HSt` environment keeps track of the serialized states of active iData elements in an iData Toolkit application. These states are either stored locally in the HTML page of the web application (in case of {n,s}[d]`IDataId` values) or reside on disk on the server side (in case of p[d]`IDataId` values). In addition, it holds a global counter to generate position values in the generic representation of state values.

```
:: *HSt      = { cntr::InputId, states::*IDataStates, world::*World }
:: InputId :== Int
```

`IDataStates` stores the serialized states of iData elements, together with their `IDataId` value, and if they have been changed by the user. `IDataStates` is basically an association list with a look-up function `lookupState` and update function `replaceState` (`replaceState` was also encountered in Sect. 3.1).

```
lookupState  :: IDataId   *HSt → (Bool,Maybe a,*HSt) | gParse{|*|} a
replaceState :: IDataId a *HSt → *HSt                | gPrint{|*|} a
```

These require the `World` environment in case of persistent forms. The generic functions `gParse` and `gPrint` are used for (de)serialization purposes.

In addition to the serialized states of iData, the `*HSt` environment also keeps track of the user modifications by storing *what* has been changed into which *new value*. This information can be retrieved by the function

```
getUserEdit :: *HSt → ((Maybe a,Maybe b),*HSt) | gParse{|*|} a & gParse{|*|} b
```

The type of `getUserEdit` reveals that we are dealing with serialized values. The first result is *what* has been changed, and the second result is its *new value*. For the identification purpose an *identification triplet* is used. Its first element is the identification string of the iData element. For convenience, it can be retrieved separately as well with

```
getIDataName :: *HSt → (String,*HSt)
```

The second element of the triplet is the value that has been changed. Generically speaking, this can only be a basic value (alternatives `UpdI` upto `UpdS`) or a data constructor (the name of which is stored in the `UpdC` alternative).

```
:: UpdValue = UpdI Int | UpdR Real | UpdB Bool | UpdS String | UpdC String
```

Of course, also the new value can be encoded in this way. The third element is the position of the generic element in the generic representation. Because the generic representation is a tree structure, this position can be obtained with a straightforward numbering scheme. This information is sufficient to determine for *any* iData element *whether* it has been changed, and, if so, *which* generic component has been changed into *what* new value. This case analysis is performed by `decodeInput`:

```
:: FormUpdate :== (InputId,UpdValue)

decodeInput :: IDataId *HSt → (Maybe FormUpdate,(Bool,Maybe a,*HSt))
             | gParse{|*|} a
decodeInput iDataId hst
   # (name,hst) = getIDataName hst
   | name == iDataId.id
      = case getUserEdit hst of
           ((Just (sid,pos,UpdI i),newi),hst) // case distinction on Int
              = let ni = case newi of (Just ni) → ni; _ → i
                in (Just (pos,UpdI ni),lookupState {iDataId & id=sid} hst)
           (_,hst) = ... // case distinction on other basic types
   | otherwise
      = (Nothing, lookupState iDataId hst)
```

This function checks whether the iData element that is identified by `IDataId` has been edited. If so, its exact location in the generic representation is returned (of type `FormUpdate`), as well as its current value (the result of using `lookupState`). It should be noted that `lookupState` may fail to parse the input (e.g. the user

entered 42.0 instead of 42 for an integer form). In that case, parsing fails, and the previous (correct) value is restored. This makes the system *type safe*.

In the previous section the pivotal function `mkIData` used the function

```
findIDataValue:: IDataId *HSt → (Bool,Maybe m,*HSt) | gParse{|*|}, gUpd{|*|} m
```

that was able to retrieve the possibly modified value of an iData identified by the `IDataId` argument. Before we can explain its definition, we need to delve into the generic function `gUpd` that is able to repair any value of type `a` to a new modified value of the same type `a`. It must be a generic function because it needs to traverse the generic data representation of the old value in order to locate the generic element that has been changed. This location is passed to the application in the identification value.

```
generic gUpd a :: UpdMode a → (UpdMode,a)

:: UpdMode = UpdSearch UpdValue InputId | UpdCreate [ConsPos] | UpdDone
```

The `UpdMode` type represents the two passes `gUpd` goes through: (`UpdSearch newv cnt`) represents the search for the generic element at location `cnt` with new value `newv`, and (`UpdCreate path`) represents the creation of new values for a selected data constructor that can be found at `path` (`:: ConsPos = ConsLeft | ConsRight`).

We illustrate the working of `gUpd` for basic types with the case for integers (the other cases for basic types are analogous):

```
gUpd{|Int|} (UpdSearch (UpdI new) 0) _ = (UpdDone,new)          1.
gUpd{|Int|} (UpdSearch val cnt)      i = (UpdSearch val (cnt-1),i)  2.
gUpd{|Int|} (UpdCreate l)            _ = (UpdCreate l,0)         3.
gUpd{|Int|} mode                     i = (mode,i)                4.
```

An existing value is replaced with `new` somewhere in a generic value at position `cnt` if `cnt` = 0, otherwise it is not changed and the position is decreased (lines 1 and 2). The default value for new integers is 0 (line 3).

The remaining code of `gUpd` proceeds polytypically except for `OBJECT`s. The generic constructor `OBJECT` marks the occurrence of a type constructor. It has access to all data constructors of that type. In this case its new value is determined by the name of the selected data constructor (`cname`). At that point, `gUpd` switches from searching mode into creation mode, in order to create arguments of the data constructor. The route to the desired data constructor is returned by `getConsPath :: GenericConsDescriptor → [ConsPos]`.

```
gUpd{|OBJECT of desc|} gUpd_obj (UpdSearch (UpdC cname) 0) (OBJECT obj)
   # (mode,obj) = gUpd_obj (UpdCreate path) obj
   = (UpdDone,OBJECT obj)
where path = getConsPath (hd [cons \\ cons ← desc.gtd_conses
                                | cons.gcd_name == cname ])
```

We now have gathered all the building blocks to explain the behavior of `findIDataValue`. As we have stated in the introduction, the key idea to the iData Toolkit is to delegate state handling to every individual iData element. Every manipulation in a web page that changes the current value of a form triggers

the execution of the Clean application on the server side. The application, and hence every iData element, has to figure out why it has been launched. There can be only three reasons: **1.** *The* iData *has no previous state.* This is the case for instance for all iData when a page is created for the first time. The iData should be initialized. **2.** *The* iData *has a previous state, but it was not edited.* This is the case when another iData has been edited. The iData should recover its previous state. **3.** *The* iData *was edited.* The application user has altered the iData. The iData should calculate its new state, given the update information and the recovered previous state. This case analysis is performed by `findIDataValue` (the numbers to the right coincide with the above cases):

```
findIDataValue :: IDataId *HSt → (Bool,Maybe m,*HSt) | gUpd{|*|}, gParse{|*|} m
findIDataValue iDataId hst
  = case decodeInput iDataId hst of
      (Just (cnt,newv),(changed,Just m,hst))                                  3.
        # m = if changed (snd (gUpd{|*|} (UpdSearch newv cnt) m)) m
        = (True, Just m,  hst)
      (_,(_,Just m,hst))                                                      2.
        = (False, Just m, hst)
      (_,(_,_,hst))                                                           1.
        = (False, Nothing,hst)
```

It uses `decodeInput` to deserialize the input data that has been passed to the web application and look for the iData element with the given identification. The reason of activating the iData element can then be determined straightforwardly.

3.3 Rendering iData

The final part of the implementation of the iData Toolkit is the rendering of iData elements into forms in such a way that forms are generated for any type, and that user manipulations can be traced back to a modified value of the same type. The key idea to realize this relationship is by associating the identification triplet (Sect. 3.2) with each element along the generic representation, and make it send the new value in case of a user action. The generic function `gForm` creates this form rendering of an iData element with a model value of type `m`:

```
generic gForm m :: IDataId m *HSt → (IData m,*HSt)
```

The basic types are handled in the same way, using the function `mkInput` and the union type `Value`:

```
gForm{|Int|} iDataId i hst
  # (form,hst) = mkInput iDataId (IV i) (UpdI i) hst
  = ({changed=False,value=i,form=[form]},hst)

:: Value = IV Int | RV Real | BV Bool | SV String | NQV String
```

The code of `mkInput` is given below. As mentioned earlier, we have used a types-as-grammar approach to specify HTML. Readers that are familiar with HTML, may be able to deduce the HTML output that is printed systematically from these algebraic data types (Sect. 3.5). We discuss the interesting parts.

```
mkInput :: IDataId Value UpdValue *HSt → (BodyTag,*HSt)
mkInput iDataId val updval hst=:{cntr}                                          1.
 = ( Input [ Inp_Type Inp_Text, Inp_Value val, Inp_Size defsize                 2.
           : case mode of                                                       3.
               Edit    = [ Inp_Name    identification_triplet                   4.
                         , `Inp_Std    [EditBoxStyle]                           5.
                         , `Inp_Events [OnChange callClean]]                    6.
               Display = [ Inp_ReadOnly ReadOnly                                7.
                         , `Inp_Std    [DisplayBoxStyle] ]] ""                  8.
   , {hst & cntr=cntr+1} )                                                      9.
where identification_triplet = encodeInfo (iDataId.id,cntr,updval)              10.
```

Basic forms in `Display` mode are read-only, and show this to the user (lines 7-8). When `Edit`ed, the web application on the server side needs to resurrected, and provided with the proper information. A script is called that sends all serialized states, the identification triplet (line 4 and 10), and the new value of the edited element back to the server, causing the application to be started with the new data (Sect. 3.2).

For the generic constructors (`UNIT`, `PAIR`, `EITHER`, `OBJECT`, and `CONS`) `gForm` proceeds polytypically. `UNIT` values are displayed as `EmptyBody`. (`PAIR a b`) values are placed below each other. (`EITHER a b`) values proceed recursively and display either their left or right value. (`OBJECT o`) values proceed recursively. The form that corresponds with (`CONS c`) values requires more HTML programming because it deals with the selection of data constructors. It generates a pull down menu which entries correspond with all data constructors. In `Edit` mode, the user can select one of these data constructors. Changes are handled in the same way as with basic types, except that the selected constructor name is passed as argument. All in all, `gForm`'s implementation requires 150 *loc*.

Finally, `gForm` has been specialized for several standard form elements. We do not discuss their implementation. They are similar to the above `Int` instance.

3.4 Handling specialization

In example 2.6, we have shown that programmers can specialize the iData Toolkit in the same way as generic functions using the function `specialize`.

```
specialize :: (IDataId a *HSt → (IData a,*HSt))
              IDataId a *HSt → (IData a,*HSt) | gUpd{|*|} a
specialize f iDataId v hst=:{cntr}
   # newIDataId      = {iDataId & id = iDataId.id<@"_"<@cntr}                  1.
   # (vF,hst)        = f newIDataId v (resetCount hst)                         2.
   # (UpdSearch _ c) = fst (gUpd{|*|} (UpdSearch (UpdI 0) -1) v)               3.
   = (vF,setCount (cntr - (c+1)) hst)                                          4.
```

It is the task of this function to embed the iData result of its argument function inside the generic representation of an arbitrary data structure. What it does is to create a new iData element that has a new `IDataId` identification value (line 1), and in which position counting starts afresh at zero (line 2). The proper new count can be derived by creatively using the functionality of `gUpd`: having

it search for an integer value at position -1 always fails, but it does return the size of the generic representation of the newly created iData (line 3). This size `c` is used to calculate the next legal position value (line 4).

3.5 Handling HTML

We have used a *types-as-grammar* approach to capture the official HTML grammar with a family of algebraic data types. We have encountered them in the above sections. The algebraic type `BodyTag` represents the collection of HTML tags from anchors (`A`) to variables (`Var`), and includes a few data constructors that allow flexible HTML generation:

```
:: BodyTag = A      [A_Attr]      [BodyTag]    | ... | Var [Std_Attr] String
           | STable [Table_Attr] [[BodyTag]] | BodyTag [BodyTag] | EmptyBody
```

One generic function, `gHpr`, has been written that generates proper HTML code from values of these types. Generic programming is not strictly necessary for this purpose. It does provide us with a concise generic function that can display any HTML code. Its core definition is only 27 *loc*. Printing of 73 types can be derived. Specialization is required for a few types, which adds 170 *loc*.

4 Discussion

In the previous section we have presented the implementation of the iData Toolkit. The implementation relies essentially on generic programming: the functions `gForm` and `gUpd` are able to manipulate values of arbitrary types in a type-safe way. The generic descriptions of these functions are small: 150 *loc* for `gForm`, and 80 *loc* for `gUpd`. We can provide specializations of these functions without changing the core definition. This greatly enhances their flexibility.

For serialization and deserialization we have used folklore generic printing and parsing functions that come with the standard generic Clean distribution. These generic functions are not essential for the iData Toolkit. Currently we are investigating whether we can use Clean dynamics for this purpose. They have as advantage that they can handle higher-order data types as well. However, their use and implementation is very delicate when compared with the robustness of their string based generic counterparts. The types-as-grammar approach of handling HTML is also very suited for generic programming. Again, it is not essential, but it has proven to provide us with concise code that is easily maintainable.

Finally, the architecture of the iData Toolkit allows us to target any arbitrary GUI library without much code modification. This is due to the fact that only state information and the change information, both in serialized form, is required by an iData Toolkit application in order to resurrect its next state and rendering.

5 Related Work

iData components are form abstractions. A pioneer project to experiment with form-based services is Mawl [4]. The <bigwig> project [8] uses Powerforms [7].

Both projects provide *templates* which, roughly speaking, are HTML pages with *holes* in which scalar data as well as lists can be plugged in (Mawl), but also other *templates* (<bigwig>). Powerforms reside on the client-side of a web application. The type system is used to filter out illegal user input. They advocate compile-time systems, just as we do, because this allows one to use type systems and other static analysis. The main differences are that in our approach *all first order user types* are admissible in iData, that iData are automatically derived from these types, and that we can use the expressiveness of the host language to obtain higher-order forms/pages.

Continuations are a natural means to structure interactive web applications. This has been done by Hughes [14], using his Arrow framework; Queinnec [23], who takes the position that continuations are at the essence of web browsers; Graunke *et al* [10], who have explored continuations as (one of three) functional compilation technique(s) to transform sequential interactive programs to CGI programs. Our approach is simpler because for every page we have a complete (set of) model value(s) that can be stored and retrieved generically in a page. An application is resurrected by recovering its previous state, merging the user modification, if any, and computing the proper next state that is re-rendered.

Many authors have worked on creating and manipulating HTML (XML) pages in a strongly typed setting. Early work is by Wallace and Runciman [26] on XML transformers in Haskell. The Haskell CGI library by Meijer [17] frees the programmer from dealing with CGI printing and parsing. Hanus uses similar types [11] in Curry. Thiemann constructs typed encodings of HTML in extended Haskell in an increasing level of precision for *valid* documents [24, 25]. XML transforming programs with GenericH∀skell has been investigated in UUXML [3]. Elsman and Larsen [9] have worked on typed representations of XML in ML [18]. Our *types-as-grammar* approach eliminates all syntactically incorrect programs, but we have not put effort in eradicating all semantically incorrect programs. Our research interest is in the automatic creation of forms from type specifications, and less in the definition of the HTML pages in which they reside.

6 Conclusions

This paper focusses on the implementation of the iData Toolkit. We have not been able to show how realistic, interconnected, real world applications are constructed with the toolkit. We have made a number of large applications, one of which is a web shop that uses many interconnected iData elements in a dynamic way, using server side data storage. Even these kind of applications can be made in the same declarative style as shown by the key toy examples in this paper.

Creating the iData Toolkit is truly a challenge because it boils down to implementing a multi-purpose unit, the iData, that automatically takes care of initialization, state recovery and update, abstraction, and rendering. Generic programming brings down the complexity significantly. It also provides us with an open-ended implementation: without modifications to the core implementation, program developers can specialize the toolkit to their own preferences and

needs per application. Although the iData Toolkit was designed for web applications, its architecture can be targeted at any graphical user interface platform without significant changes. This is a major improvement to our previous work on desktop applications. The implementation is concise, elegant, and efficient. In all, the results of this project show that the iData Toolkit is an excellent case study in the appropriateness of generic programming.

Acknowledgements

Jan Kuper coined the name iData for our editor components. Pieter Koopman provided input for the gUpd function. Paul de Mast kindly provided us with a web server application written in Clean which has allowed us to readily test the iData Toolkit. Javier Pomer Tendillo, as an Erasmus guest, has been helpful in setting up the toolkit, and find out the nitty-gritty details of HTML programming. Finally, we thank the anonymous referees.

References

1. A. Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, University of Nijmegen, The Netherlands, 2005. ISBN 3-540-67658-9.
2. A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, Sept. 2002.
3. F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema-Haskell Data Binding. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *LNCS*, pages 71–85. Springer-Verlag, June 2004.
4. D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a Domain Specific Language for Form-based Services. In *Usenix Conference on Domain Specific Languages*, Oct. 1997.
5. E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, volume 6, pages 579–612, 1996.
6. E. Barendsen and S. Smetsers. *Graph Rewriting Aspects of Functional Programming*, chapter 2, pages 63–102. World scientific, 1999.
7. C. Brabrand, A. Møller, M. Ricky, and M. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4):205–314, 2000.
8. C. Brabrand, A. Møller, and M. Schwartzbach. The <bigwig> Project. In *ACM Transactions on Internet Technology (TOIT)*, 2002.
9. M. Elsman and K. F. Larsen. Typing XHTML Web applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, volume 3057 of *LNCS*, pages 224–238. Springer-Verlag, June 2004.
10. P. Graunke, S. Krishnamurthi, R. Bruce Findler, and M. Felleisen. Automatically Restructuring Programs for the Web. In M. Feather and M. Goedicke, editors, *Proceedings 16th IEEE International Conference on Automated Software Engineering (ASE'01)*. IEEE CS Press, Sept. 2001.

11. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.
12. R. Hinze. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.
13. R. Hinze and S. Peyton Jones. Derivable Type Classes. In G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop*, volume 41(1) of *ENTCS*. Montreal, Canada, Elsevier Science, 2001.
14. J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.
15. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.
16. A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 141–152. ACM Press, 2003.
17. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.
18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
19. S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.
20. R. Plasmeijer and P. Achten. Generic Editors for the World Wide Web. In *Central-European Functional Programming School*, Eötvös Loránd University, Budapest, Hungary, Jul 4-16 2005.
21. R. Plasmeijer and P. Achten. iData For The World Wide Web - Programming Interconnected Web Forms. In *Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *LNCS*, Fuji Susono, Japan, Apr 24-26 2006. Springer Verlag.
22. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.
23. C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings Fifth International Conference on Functional Programming (ICFP'00)*, Sept. 2000.
24. P. Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In S. Krishnamurthi and C. Ramakrishnan, editors, *Practical Aspects of Declarative Languages: 4th International Symposium, PADL 2002*, volume 2257 of *LNCS*, pages 192–208, Portland, OR, USA, January 19-20 2002. Springer-Verlag.
25. P. Thiemann. A Typed Representation for HTML and XML Documents in Haskell. *Journal of Functional Programming*, 2005. Under consideration for publication.
26. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the Fourth ACM SIGPLAN Intnl. Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 1999. ACM.